Homework 12 Introduction to Big Data Systems

Christoffer Brevik

December 9, 2024

1 Code Overview

For this week's assignment, I have implemented a dilated convolution algorithm in C++ using the Halide library. The program computes the output of a dilated convolution operation for a given set of inputs and filters, then benchmarks my codes performance against oneDNN for comparison.

1.1 Dependencies

The implementation relies on the following dependencies:

- **Halide** - A language for image processing and computation, used to define and optimize the dilated convolution operation.
- **oneDNN** - An open-source performance library for deep learning applications, used to benchmark the results and performance of the Halide implementation.
- **Common Utilities** - Includes helper functions for random data generation, benchmarking, and result verification.

All dependencies are installed and accessible on the computational node, ensuring the program runs without additional setup.

1.2 How to Run the Code

To execute the program and evaluate the results, open a terminal and navigate to the folder containing the source files:

```
cd /path/to/source
```

Then, compile the program using the provided `Makefile`:

```
make dilated_conv
```

With the file sucessfully compiled, we can run the executable with the command:

```
srun -n 1 -c 4 ./dilated_conv
```

The code will now start, and the program will display it's result in the console. It will display:

- Performance metrics for the Halide implementation, including execution time and GFLOP/s.
- Performance metrics for the oneDNN implementation for comparison.
- A correctness check indicating whether the Halide results match the oneDNN reference results.

Upon completion, the program should output `Success!` to confirm that my code gives the correct execution.

2 Code Implementation

The program implements a dilated convolution algorithm using the Halide library. As tasked in the assignment, I have modified the provided `dilated_conv.cpp` instead of making any new files to the project. As most of the code was more or less unchanged, I will not explain how the data is set up, the code is run or how my code is compared. What I will focus on is the scheduling and how this is implemented to make my code as fast as possible:

2.1 Scheduling Strategy

The heart of the optimization lies in the application of Halide's scheduling directives to ensure efficient computation and memory usage. The following schedule was implemented:

```
output.tile(c, x, c_out, x_out, c_in, x_in, 64, 4)
      .vectorize(c_in)
      .vectorize(x_in)
      .fuse(c_out, x_out, tile_idx)
      .parallel(tile_idx)
      .parallel(y)
      .parallel(n);
```

The scheduling does the following

Tiling

- `tile(c, x, c_out, x_out, c_in, x_in, 64, 4)`: This breaks the computation into smaller tiles of size 64×4, effectively splitting the output tensor into manageable chunks.

Vectorization

- `vectorize(c_in)` and `vectorize(x_in)`: These directives enable vectorized computation along the innermost dimensions of the tiles (`c_in` and `x_in`).

Parallelization

- `fuse(c_out, x_out, tile_idx)`: Combines the outer tile indices into a single loop variable (`tile_idx`) for parallel execution.

- `parallel(tile_idx)`, `parallel(y)`, and `parallel(n)`: These directives distribute computation across multiple threads for the fused tile indices (`tile_idx`), along the y dimension, and across batch processing (`n`).

3 Results

3.1 Performance Metrics

To evaluate the performance of my implementation, I compared the execution time and throughput (measured in GFLOP/s) of my optimized Halide solution with oneDNN's implementation. The results are as follows:

- **(My) Halide Results:** Execution time of 150.08 ms with a throughput of 78.60 GFLOP/s.
- **oneDNN Results:** Execution time of 51.03 ms with a throughput of 231.17 GFLOP/s.

3.2 Console Output

The Halide implementation produced the expected output, verifying its correctness. The console confirmed successful execution with the message:

```
Halide results - OK

[...]

Success!
```

3.3 Comparison of Results

A direct comparison of performance metrics is summarized in Table 1:

Dilation	Halide Runtime	Halide Throughput	oneDNN Runtime	oneDNN Throughput
0	147.96	79.72	45.99	256.46
15	147.90	79.75	46.15	255.60
31	146.40	80.57	46.17	255.49
63	147.51	79.96	46.16	255.54

Table 1: Performance metrics for Halide and oneDNN

3.4 Conclusion

While the Halide implementation is slower than oneDNN, achieving approximately 34% of its throughput, it was still severely better than the example given in the assignment. Therefore I still believe that my code is good. I did try multiple other strategies, but this proved to be the fastest solution.